USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Offloading Cloud Network Services at Production Scale with SONiC DASH SmartSwitch

Shaofeng Wu, *The Chinese University of Hong Kong and Microsoft Research Asia;* Zhixiong Niu, *Microsoft Research Asia;* Riff Jiang, Lawrence Lee, Junhua Zhai, Ze Gan, Vasundhara Volam, Prabhat Aravind, Prince Sunny, Prince George, Qi Luo, Evan Langlais, Soumya Tiwari, Venkat Satish Katta, Weixi Chen, Rishiraj Hazarika, Sachin Jain, Deven Jagasia, Michal Zygmunt, Avijit Gupta, Neeraj Motwani, and Pranjal Shrivastava, *Microsoft;* Qiang Su, *The Chinese University of Hong Kong;* Anil Reddy Pannala, Kristina Moore, James Grantham, Anupam Pandey, Xin Liu, Guohan Lu, Gerald De Grace, Rishabh Tewari, Lihua Yuan, Erica Lan, Deepak Bansal, and Dave Maltz, *Microsoft;* Yongqiang Xiong, *Microsoft Research Asia;* Hong Xu, *The Chinese University of Hong Kong*

https://www.usenix.org/conference/nsdi26/presentation/wu-shaofeng

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Offloading Cloud Network Services at Production Scale with SONiC DASH SmartSwitch

Shaofeng Wu[1,2], Zhixiong Niu[2*], Riff Jiang[3], Lawrence Lee[3], Junhua Zhai[3], Ze Gan[3], Vasundhara Volam[3], Prabhat Aravind[3], Prince Sunny[3], Prince George[3], Qi Luo[3], Evan Langlais[3], Soumya Tiwari[3], Venkat Satish Katta[3], Weixi Chen[3], Rishiraj Hazarika[3], Sachin Jain[3], Deven Jagasia[3], Michal Zygmunt[3], Avijit Gupta[3], Neeraj Motwani[3], Pranjal Shrivastava[3], Qiang Su[1], Anil Reddy Pannala[3], Kristina Moore[3], James Grantham[3], Anupam Pandey[3], Xin Liu[3], Guohan Lu[3], Gerald De Grace[3], Rishabh Tewari[3], Lihua Yuan[3], Erica Lan[3], Deepak Bansal[3], Dave Maltz[3], Yongqiang Xiong[2], Hong Xu[1]

[1]*The Chinese University of Hong Kong,* [2]*Microsoft Research Asia,* [3]*Microsoft*

Abstract

To support stateful cloud network services, Microsoft Azure has operated several generations of offloading solutions over the past decade. While these systems improved performance, operating them at hyperscale revealed three persistent lessons: (i) overly flexible programming models hinder hardware acceleration, (ii) appliance-style DPU pools inflate physical footprint and complicate deployment, and (iii) vendor-specific SDKs slow down service iteration.

We present SONiC DASH SmartSwitch that addresses these lessons with three key designs: (1) the DASH pipeline as an immutable and hardware-friendly programming model; (2) the uni-box SmartSwitch that converges NPU and DPU resources within a single T1 switch; and (3) a community-driven development model with P4 behavior specifications. SONiC DASH SmartSwitch has been deployed in Microsoft Azure at scale. It achieves 1.53Tbps throughput, 19.2M CPS, and 256M concurrent connections for network services, while improving power efficiency by ∼1.8× and space efficiency by ∼2.7× compared to the previous generation.

1 Introduction

Modern public clouds must provide a portfolio of *stateful* network services — virtual networking (VNET) [9,29,30,33–35,43], load balancing [5,45], NAT gateways [2,6], and private connectivity [7,8] — that deliver isolation, policy, and reachability at a massive scale. These services must sustain high throughput and packet rates, rapid connection setup (CPS), and hundreds of millions of concurrent flows. In Azure, these metrics are critical: throughput and concurrent connections bound tenant scale and workload density; CPS dictates how fast new VMs and services can come online [11,14,30,38].

Much work has explored hardware offloading to accelerate stateful network services [36–41,43–45], and some systems are deployed in production [38,40,43,45]. At Microsoft Azure, we have operated and evolved several generations of service offload architectures over the past decade (Table 1). The host-SDN design VFP [34] offered unmatched flexibility but proved too costly in CPU consumption. AccelNet [35] and Bluebird [29] improved throughput by offloading critical paths to FPGAs or NPUs,[1] yet soon encountered scaling ceilings and fragmented the programming interface. Most recently, Sirius [30] disaggregated network services onto DPU pools to overcome single-device limits. On the other hand, its reliance on vendor-specific SDKs, and beefy appliance-style racks with auxiliary servers and extra optics and switches, continue to strain its deployment.

Our hyperscale deployment experiences teach us several important lessons as we set out to design the next generation offloading solution. *(i) Programming flexibility can be curbed for maximum hardware acceleration.* In particular, we observe that the unbounded match-action models from VFP resist line-rate optimization and cap CPS; they can be removed without sacrificing expressiveness in implementing services. *(ii) Physical footprint matters.* Appliance-style resource pools like Sirius introduce extra hardware, power and rack space, inflating both CAPEX and OPEX and limiting deployability. *(iii) Vendor-centric development slows down feature iteration.* Relying on vendor SDKs for performance comes at a high cost of iteration velocity and supply risks.

This paper presents SONiC DASH SmartSwitch, a software-hardware co-design for hyperscale cloud network service offloading. SONiC DASH SmartSwitch combines (1) **DASH pipeline**, an *immutable* yet configurable abstraction that captures recurring service requirements while enabling efficient line-rate implementations; (2) **SmartSwitch**, a uni-box T1 switch that co-locates NPU and DPU resources to reduce auxiliary equipment and footprint; and (3) a **community-driven development workflow** that uses P4 behavior models as executable ground truth to enable portable, multi-vendor implementations.

We implement SONiC DASH SmartSwitch in Azure, and it delivers line-rate forwarding (1.53Tbps, 19.2M CPS, 256M

*Corresponding author: Zhixiong Niu (zhniu@microsoft.com).

[1]In this paper, NPU (Network Processing Unit) refers to a programmable switch chip, while DPU denotes a Data Processing Unit. All other acronyms in this paper are listed in Appendix §A and will not be explained separately.

concurrent connections) and improves power/space efficiency by ∼1.8×/∼2.7× over the prior generation. Over 4 years, DASH [14] has attracted hundreds of contributors across Microsoft, AMD, Cisco, Nvidia, Keysight, and others, enabling multi-vendor collaboration for hyperscale cloud networking.

Our key contributions are as follows:

- We, for the first time, share with the community three key operational lessons from real deployments of cloud network services, regarding the issues of over-flexible programming model, costs of physical footprint of the hardware, and risks of vendor-centric development (§2).
- We design and build SONiC DASH SmartSwitch, featuring DASH pipeline as the programming model (§4), SmartSwitch as the hardware model (§5), and a community-driven open development model (§6).
- We present evaluation results and deployment lessons about SONiC DASH SmartSwitch that shed light on future research (§7, §8).

2 Motivating SONiC DASH SmartSwitch

Over the last 15 years, we have operated and evolved several generations of cloud network services solutions in Microsoft Azure [29, 30, 34, 35, 39], as summarized in Table 1. Our first solution, VFP [34], is a software-based host SDN approach. It runs as a virtual switch in the hypervisor and provides high flexibility to implement different network services. As software-based approach cannot meet the performance requirements of emerging workloads, we moved to hardware offloads with AccelNet [35], which offloads VFP's fast path to FPGAs on each host. We also leveraged top-of-rack (ToR) programmable switches to offload VNET-to-VNET service for bare-metal servers in Bluebird [29]. Essentially, both AccelNet and Bluebird are still limited by high CAPEX and resource limitations of locally connected programmable devices (FPGA-based SmartNICs and switches, respectively). To fundamentally overcome this limitation, Sirius [30] disaggregates stateful network functions onto a DPU resource pool attached to T1 switches, representing the latest generation of *disaggregated* offloading solutions.

SONiC DASH SmartSwitch is motivated by three unique lessons we learned through this journey, covering three key dimensions: (1) programming model, *i.e.,* the abstractions, APIs, and semantics for SDN controller to specify service logic and policies, (2) hardware model, *i.e.,* how physical and logical hardware resources (e.g., NPUs, DPUs, SRAM/-DRAM, and interconnects) are architected and presented to the software stack, and (3) development model, *i.e.,* the business, governance, and engineering processes by which cloud providers and hardware technology providers collaborate to specify, implement, test, and deploy services. We share these lessons for the first time and believe that they will be useful to the wider research community.

Lesson I: Flexible programming model is NOT (always) necessary.

Flexibility v.s. Performance. A critical lesson we learned when moving from the software-based approach to hardware offloads for higher performance is that the choice of programming model inevitably involves trade-offs between flexibility and performance: when the model is flexible, it is non-trivial or even impossible for hardware providers to implement and optimize their hardware accordingly due to resource and semantic constraints of commodity hardware.

VFP is designed to support very flexible match-action rules in its layers.[2] A single rule supports a list of matching conditions that have a type (such as source IP address) and a list of matching values (each value may be a singleton, range, or prefix). The consequence is that layer rules form a huge, unbounded <condition, action> space, and hardware cannot assume a small, fixed set of conditions or actions to accelerate. In fact, we only successfully offloaded VFP's fast path to FPGA in AccelNet [35] and leave the complex layer rules (and other software-native functionalities, *e.g.,* unbounded layers, rich callbacks that support arbitrary actions, *etc.*) to host CPU, because commodity FPGA and ASIC pipelines are incapable of processing large amounts of customized rules with heterogeneous match fields and types at line rate. This results in a low connection rate (O(100K) CPS per core) and fewer available cores for user VMs. To meet production performance requirements, we opted to use proprietary vendor SDKs instead of VFP's flexible model to program our network services in later generations of hardware solutions: Bluebird [29] and Sirius [30]. Vendor SDKs are well-optimized on their platforms, but their programming models do not interoperate, which defeats another goal of multi-sourcing (Lesson III). P4 is also a good fit for programming P4-programmable switches, but they are usually used together with proprietary SDKs [3] or are even not applicable on other platforms such as DPUs [18, 19].

Flexibility v.s. L4 service requirements. We also find that service scenarios in Azure and other hyperscale clouds, to the best of our knowledge, typically require only a subset of the rich features provided by VFP [1, 4–6, 8–10, 16, 27, 35], and each feature can be shared across many services. For example, all of our services require state tracking at per-flow or per-endpoint granularity; variants of virtualized and private connection services in Azure (VNET-to-VNET, Private Link, VNET Peering) maintain CA-PA mappings and VxLAN routing, which are also fundamental to similar services in other clouds [1, 43]. Although flexible programming models such as VFP and P4 support these features, they are often "overkills" — their excessive flexibility introduces compatibility and interoperability issues for vendors. In practice, abstracting these capabilities as functional modules and reusing them provides a more reasonable and effective approach for constructing cloud network services.

[2] VFP's layers refer to stateful flow tables that hold match-action policies. Packets go through each layer one by one, matching rules in each based on their state after the action performed in the previous layer.

	VFP [34]	AccelNet [35]	Bluebird [29]	Sirius [30]	SONiC DASH SmartSwitch
Hardware Model	Host-based	FPGA-based	NPU-centric	DPU-centric	Unibox NPU+DPU
Throughput (pps)	O(10M)/core	Up to 200Mpps	5-7Bpps	O(100Mpps)	
Throughput (bps)	<40Gbps/core	Up to 200Gbps	6.4-12.8Tbps	O(1Tbps)	
CPS	O(100K)/server core, expensive to scale-up		O(1M), hard to scale-up	O(10M)	
# Connections	>100M	O(10M)	O(1M)	O(100M)	
Cost	High, consumes server cores	High, per server and high CAPEX	Low	Medium, shared pool	Low, highly-converged shared pool
Rack Footprint	Low	Low	Low	High, extra space required	Low
Multi-sourced?	N/A	✗	✗	✗	✓

Table 1: Evolution and comparison of cloud network service solutions in Azure.

Design objective I. We need a compact and vendor-neutral programming model that provides just enough flexibility for L4 services while being hardware friendly (§4), especially for those critical services whose performance must be prioritized over flexibility.

Lesson II: Physical footprint matters.

Another important lesson concerns the physical footprint of the hardware models of our solution and their implications on space occupancy, power consumption, and operating cost, which unfortunately have received little attention in the research community. We elaborate this lesson with our latest solution Sirius [30], which disaggregated a network service onto a DPU pool to overcome the performance limitations of offloading to a single device. However, through operating it in Azure we observe that its deployment is not as efficient and scalable as we ideally expect.

Excessive auxiliary equipment. First, Sirius contains much auxiliary equipment. As shown in Figure 1, in addition to 2×6 DPUs, each Sirius appliance entails the new installation of 2 servers (2×3RU) for cabling, cooling and powering, and 2 ToR-equivalent switches (2×1RU) for basic connectivity between the appliance and datacenter core and between the pair of servers. None of this equipment directly contributes to stateful packet/flow processing but does contribute to CAPEX, power consumption, floor space occupancy and human-hours. Our analysis in §7.3 shows we can reduce the power consumption and rack space of a Sirius appliance by 365.82W and 2RU with 33% higher throughput/CPS/number of connections and lower latency.

Massive floor space footprint. In our deployments, a separate and purpose-built rack is needed to accommodate the beefy Sirius appliance, including its DPU-attached servers and switching system. We initially chose to dock this new rack to rows connected by shared converged middle-of-rack (MoR) switch in the public preview stage. These rows are less populated and have empty space that allows us to directly place the Sirius appliance and deploy more racks at a fast pace. However, such rows are limited in a datacenter and we were only able to deploy tens of Sirius appliances in each Azure zone, leaving most compute racks out of acceleration.

As we move from public preview to general availability, we pivot to installing Sirius racks alongside pre-built compute rows that house compute nodes and account for the majority of rack capacity in our datacenters. Figure 1 shows the deployment position of Sirius appliances: since each compute row is pre-designed to house 2k+1 racks (2k for servers and ToRs, 1 for MoR switch), which have already been fully occupied by servers, switches and other rack equipment, we have no choice but to use an extra tile of floor space and misappropriate Sirius appliances to the aisle next to T1 rows (as the 2k+2-th rack of each row). This severely limits scalability and deployability because floor space and cooling capacity are pre-designed and should not be forcefully surpassed considering the negative impact on circulation, safety and maintenance difficulty for human operators [42]. Another choice is to remove one server rack per row to make floor space for Sirius appliances, which, however, comes at the cost of reduced VMs for sales.

Other factors. In addition to the above issues, other factors also limit the deployability of Sirius. For example, because the DPU-centric appliance is a separate rack and has its own support lifetime that is different from that of other equipment, *e.g.,* servers and switches, it is deployed and decommissioned with new procedures, which increases OPEX.

Design objective II. "A design that scales only on paper, and not in physical reality, is not truly scalable" [42]. Physical footprints of hardware are a top concern in Azure as real deployments are always under physical constraints regarding space, power, etc. This necessitates a new hardware model with a much smaller space and power footprint compared to Sirius. Compatibility with existing infrastructure and standard procedures of datacenters should also be taken into account to reduce CAPEX and OPEX.

Lesson III: Software development model is important for service iteration.

While the software development model may not seem relevant for research prototypes or small-scale clouds, it matters for production-scale deployments in hyperscale clouds because it determines whether and how efficiently cloud network services can be iterated on a heterogeneous hardware fleet to reduce time-to-market and support business growth [11, 35].

So far we have followed a *vendor-centric* development model: building our services with proprietary SDKs for hardware offloads and letting hardware vendors decide which features are available to us. For example, services offloaded to Sirius appliances are actually built by a team from the hardware vendor with their proprietary SDK. We have since found it difficult to evolve our services timely and flexibly enough. This can be attributed to three key limitations of the develop-

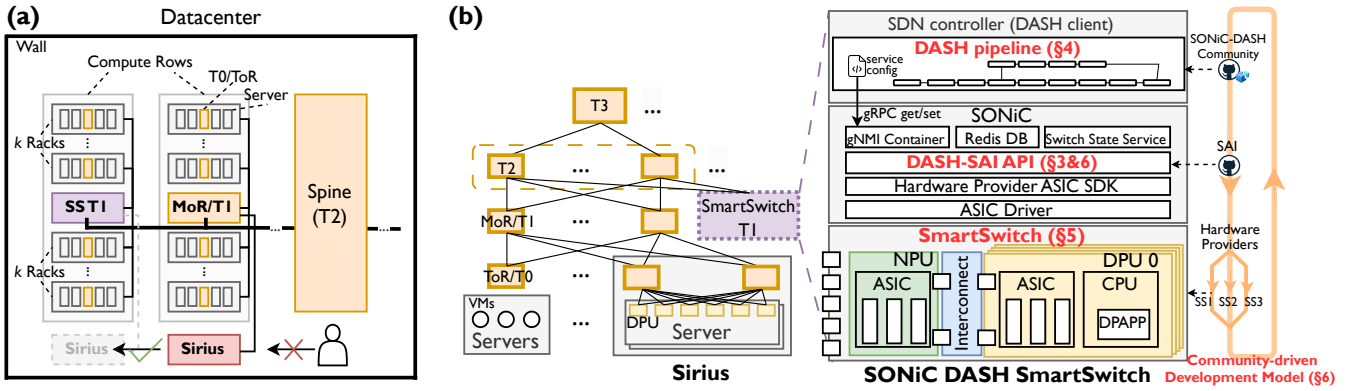

Figure 1: (a) Physical layout of a datacenter (in elevation view) comparing the deployment footprint of SmartSwitch (SS) and Sirius. (b) Logical diagram of the architecture of DPU-centric appliance (Sirius) and SONiC DASH SmartSwitch.

ment model. (1) Roll-outs of vendor-proprietary SDKs are slow for our purposes. A major release of SDKs from our collaborators takes 6–12 months on average. (2) Vendor SDKs are not service-oriented and may diverge from our feature requirements. (3) Services developed with SDKs are black boxes for us and there is no single source of truth that clearly specifies their behaviors. For example, a "Region" field in packets needs to be changed for ACL checks in the Private Link service of Sirius, but the documentation provided by the vendor does not clearly specify this behavior and can be misleading if we want to adjust the behavior of existing services or deploy new ones. In addition, we are also exposed to supply chain risks and cannot ensure business continuity under the current vendor-centric model because hardware providers may cease their support of a product [40], *e.g.,* Tofino [17] used by our previous solution, Bluebird [29], and mature code for control plane, data plane and testing cannot be directly re-used on new platforms due to incompatible interfaces.

Design objective III. Our new solution needs to follow a multi-sourced development model to ensure business continuity and resilience. Specifically, we should pay special attention to avoiding black-boxed service behaviors caused by vendor-specific SDKs, and provide a mechanism for *white-box modeling* of service features and behavior so that we can rapidly *propose, prototype, test and deploy* new features according to the latest requirements from our clouds without being blocked by vendor-specific details.

3 SONiC DASH SmartSwitch Overview

Before we initiated SONiC DASH SmartSwitch project in 2021, we had widely deployed SONiC-based switches in Azure due to the production-ready stack and open source ecosystem of SONiC [26]. SONiC uses the Switch Abstraction Interface (SAI) [25], a set of low-level vendor-neutral APIs, to configure and control different forwarding elements (switching ASIC, NPU, software switch, etc.) in a uniform manner. SAI focuses on L2 and L3 stateless forwarding features, *e.g.,* IP multicast, (stateless) ACL, ECMP, etc., but lacks L4 stateful abstractions required by cloud network services.

DASH — Disaggregated APIs for SONiC Hosts. When we set out to build our new service offload solution with goals in §2, we naturally use SONiC and SAI as our starting point to reuse their mature code and keep vendor-neutrality. To support cloud network services, we extend SAI with new headers containing L4 stateful APIs and object models, *e.g.,* flow table and table/entry operations. These new headers are named Disaggregated APIs for SONiC Hosts, DASH.

However, DASH and SAI function merely as a set of vendor-neutral API standards; on their own, they cannot address the fundamental operational challenges in §2. Upon DASH, we build SONiC DASH SmartSwitch, a multi-sourced, performant and efficient software-hardware co-design for offloading cloud network services at scale. Figure 1(b) shows its overall architecture. We make three major design contributions.

Programming model — DASH pipeline (§4). SONiC DASH SmartSwitch adopts *DASH pipeline*, a match-action table (MAT) pipeline for representing cloud network services logic. DASH pipeline has a fixed set of immutable stages, each implementing a common functionality such as connection tracking, stateful ACL, etc. We purposely limit the flexibility of DASH pipeline and only allow adjusting the data-plane logic by configuring a set of exposed stage attributes. For example, we can specify the flow key fields used by the flow table (in a pipeline stage), but we cannot change the pre-defined match type from exact match to others, which is allowed by VFP. To program a service, SDN controller developers write configurations following DASH pipeline model and SONiC will invoke the corresponding DASH APIs to program service logic into ASIC. In our production deployments, DASH pipeline provides just enough flexibility to represent our services because they share a lot of functionalities, and achieves high performance simultaneously since the pre-defined data plane is easy to implement and optimize on hardware.

Uni-box hardware model — SmartSwitch (§5). Our hardware model is a *unibox* switch that converges an NPU and multiple DPUs to achieve performance, efficiency and deployability simultaneously. SmartSwitches are deployed as T1 switches and services are disaggregated to their DPU resource

pool, processing stateful traffic directly on the datapath without detouring to Sirius appliances. DASH is designed to work on DPUs and service logic is programmed to SmartSwitch DPUs to support scalable throughput and number of connections at low space/power/cost footprints.

Community-driven development model (§6). The development of SONiC DASH SmartSwitch software and hardware is under a new community-driven development model. An open community, including both cloud providers and hardware technology providers, jointly decides which features are required by public clouds and should be supported by DASH, DASH pipeline and SmartSwitch. We enable this process by modeling the standard behavior of DASH pipeline stages with P4 sources, *a.k.a.* behavior model. P4 sources are used as the community agreement with no ambiguity; once the community reaches a consensus on a new feature, it is introduced as a new stage with its behavior documented as P4 sources and merged into the existing behavior model. We also use the behavior model to automate the workflow, *e.g.,* generating DASH headers from P4 sources and running sources on a software switch to test before hardware becomes available.

4 DASH Pipeline

Our first contribution is a high-level programming model for cloud network services, which is named as DASH pipeline because it is mapped to hardware via lower-level DASH-SAI APIs. DASH pipeline is designed to work as a general-purpose and future-proof pipeline for programming cloud network services in Azure and other public clouds, while being friendly to hardware offloads and optimizations.

The architecture of DASH pipeline is shown in Figure 2. Our guiding design principle here is to retain from VFP those L4 primitives that have been validated over the years in production, while eliminating its vast, unbounded <condition, action> space and constraining programmability to the minimum necessary to meet Azure's production performance targets. The principle is realized by defining services with a *unified, immutable* and *configurable* MAT pipeline.

4.1 Stages

DASH pipeline can be logically split to multiple functionality units, referred to as *stages*. Each stage implements a subset of cloud network service logic with one or multiple MATs. For example, `ConnTrack Lookup` stage implements a flow table (a MAT with five tuples as key) that stores <flow key, flow state> tuples and tracks per-flow states.

Immutability. A stage is conceptually similar to a VFP layer in that they both represent a set of stateful MATs. The key difference is that a stage is *immutable*. The immutability has five aspects:

1. Number. The number of stages is pre-defined in DASH pipeline; in contrast, an SDN controller may create an arbitrary number of layers in VFP.

2. Topology. Stages are connected in a fixed topology. Packets are processed sequentially in a pre-defined order with possible conditional branching unless they are dropped by a stage.

3. Entry Shape. A stage has a fixed number of MATs. Entries of a given MAT have fixed key/value shapes, meaning that (1) the number of fields, the type of each field (*e.g.,* 8-bit or 16-bit), the source of a field (where to obtain the value from, *e.g.,* source IP) and the match type (*e.g.,* exact match or longest-prefix match) in entry key, and (2) the number and type of fields in entry value (action parameters, pointers, counters, *etc.*) are pre-defined. Entry shape determines what functionality a stage implements. For example, the entry key of the flow table in `ConnTrack Lookup` can only be (`VNI`,`ENI MAC`,`src IP`,`dst IP`,`src Port`,`dst Port`,`Protocol`) or its subset. Fields of entry key can be disabled when only a subset is needed and all entries in an MAT must have the same shape; in contrast, rules in a VFP layer may have heterogeneous shapes.

4. Action. A stage performs pre-defined actions. For example, if flow lookup hits an entry, the action in `ConnTrack Lookup` is simply publishing all fields in entry value to the metadata bus.[3] There are some stages that do not use MATs; they perform actions based on pre-defined logic (see §4.2).

5. Stage Attribute. A stage exposes a fixed set of configurable attributes to allow controlled customizations. For the mentioned example of disabling entry key fields, it can be configured in `ConnTrack Lookup` stage by setting the value of `ENABLED_KEY`, which is a key mask and applies to all entries in the flow table.

Immutable stage definitions make it straightforward for vendors to map DASH pipeline objects to hardware components and optimize their implementations. A vendor can use SRAM and DRAM to store the large flow table in `ConnTrack Lookup`, and use TCAM for list match and range match in `Pre-/Post-pipeline ACL`, and hardware resource budgeting on specific targets can be smoothly achieved.

Supported stages. We distilled the 13 stages in the current DASH pipeline from the common requirements of Azure's and other major clouds' mission-critical network services. For example, a majority of our network services require parallel and isolated context, *e.g.,* virtual NICs of different tenants' VMs, which provide VNET-to-VNET, and load balancer instances with different policies for different backend resources. Accordingly, we introduced a `Pipeline Lookup` stage to dispatch traffic to parallel DASH pipelines based on configurable identifiers. Another example is that Azure VMs typically require two sets of separate network security policies, *i.e.,* customer and infrastructure network security policies, which drives us to design two ACL stages (`Pre-/Post-pipeline ACL`). Due to page limits, we describe the purpose and use cases of other stages in Appendix §B.

[3] Metadata bus is a common feature of ASIC pipeline to provide per-packet context, *e.g.,* packet header vector (PHV). We provide its abstraction in DASH pipeline and define a fixed set of available fields. Each stage may read or write metadata bus fields.

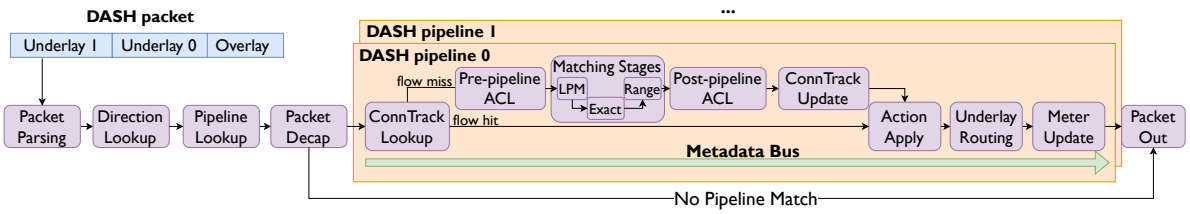

Figure 2: Architecture of DASH pipeline. Each DASH pipeline represents a pipeline instance, *e.g.,* an ENI for VNET-to-VNET, and some stages are shared by pipelines, *e.g.,* `Pipeline Lookup`.

Service	Functionality & Main Operations	Pipeline Lookup	Packet Decap	ConnTrack Lookup	Action Apply
VNET-to-VNET [9]	VM to VM communication in virtual network. CA-PA mapping and per-flow packet encap. & decap..	{inner smac}, (inner dmac)	✓	VNI, 5-tuple	encap_u0, set_dmac
VNET Peering [10]	Connectivity between virtual networks within the same Azure region or across Azure regions. CA-PA mapping and per-flow packet encap. & decap..	{inner smac}, (inner dmac)	✓	VNI, 5-tuple	encap_u0, set_dmac
Private Link [7, 8]	Private connectivity from a virtual network to Azure platform as a service (PaaS), *e.g.,* Azure Storage, and Azure hosted customer-owned/partner services. CA-PA mapping, IPv4-IPv6 address translation and per-flow packet encap. & decap. (NVGRE).	{inner smac}, (inner dmac)	✓	VNI, 5-tuple	{nat46}, (nat64), {encap_u1}, encap_u0, set_dmac
Load Balancer [5, 45]	Load balanced connectivity from VMs to Azure services or VMs. Mapping flow with destination VIP to backend server(s).	(dip, dport)	{✓}	5-tuple	(dnat), (encap_u0), (set_dmac)
NAT Gateway [2, 6]	Private connectivity between VM and Internet. Replacing private source IP & port (VM) with public ones.		{✓}	{VNI}, 5-tuple	{snat}, (dnat), (encap_u0), (set_dmac)

Table 2: Azure services implemented with DASH pipeline and their key behaviors. "Packet Parsing", "Direction Lookup", "Matching stages", "ConnTrack Update", "Underlay Routing" and "Packet Out" are omitted since all five services must use them. "ACL" and "Meter Update" are optional and they are also omitted in this table. "{}" and "()" represent the configuration is only valid for outbound (from VM) and inbound (to VM) direction respectively. "Pipeline Lookup" column and "ConnTrack Lookup" column show the key fields configured for matching pipeline and flow respectively. "Action Apply" column shows the action to be applied on packets.

Azure service scenarios. DASH pipeline is a *configurable* data-plane model. An SDN controller (a DASH client) programs a service by configuring stage attributes and MAT entries, assuming the hardware target implements the DASH pipeline semantics. SONiC acts as the control-plane adaptor that exposes unified northbound configuration endpoints to SDN controllers, persists controller objects in its databases, and translates controller objects into low-level DASH-SAI API calls that actually configure the device.[4] For user VMs and bare-metals, service capacity can be allocated by configuring the `cps`, `pps` and `number of flows` of the pipeline (an entry in the MAT of `Pipeline Lookup` stage).

Table 2 lists part of the configurations of 5 key Azure services. DASH pipeline works well for defining current workloads in Azure. It can also be extended to support future services and meet specific needs of other public clouds, and we discuss our efforts on the community-driven development model to achieve the extension in §6. For the rest of this section, we present other important components of DASH pipeline and our design choices.

4.2 Packet Parser

The first stage of DASH pipeline is `Packet Parsing`, which parses the packet header and writes encapsulation information into metadata bus. It does not use MATs, and hardware technology providers are expected to map it to ASIC parser implementations, *e.g.,* programmable finite state machine. As shown in Figure 2, we limit the packet supported by DASH to have at most two layers of encapsulation (underlay0 and underlay1) to avoid the pipeline potentially experiencing unexpected latency. The packet structure is also designed to satisfy virtualization requirements in Azure: underlay0 is used for implementing common virtual network functions, *e.g.,* VxLAN [23] in VNET-to-VNET, NVGRE [24] in Private Link (PL), *etc.*, and underlay1 is used for additional routing hops as required by many Azure services, *e.g.,* tunneling VM-to-PL traffic to a Network Security Group [7, 8] appliance.

[4] Details of SONiC DASH integrations are out of the scope of this paper.

4.3 Slow Path Policy

DASH pipeline segments datapath to fast path and slow path to align with the processing pattern of Azure services [6, 8–10] and to achieve high PPS and CPS. The mechanism is similar to VFP's: the first packet of a flow is examined against policies programmed in slow path to derive what actions should be conducted on following packets of the same flow, and the derived flow entry (flow key, actions and other states) is inserted into the flow table of `ConnTrack Lookup` stage.

Slow path policies can be complex and varied for different services. For example, VNET-to-VNET translates customer address (CA) to and from provider address (PA) with cascaded longest-prefix match and exact match on the overlay destination IP address, which differs from NAT Gateway's source IP and port mapping to a new source IP and port by exact match. VFP's flexible rules do support such customized slow path policies, but they can be hard to implement and optimize with hardware for high CPS. Therefore, we pre-define a set of fixed pipeline profiles for slow path of DASH pipeline. Specifically, a profile connects three types of `Matching Stages` that conduct longest-prefix match on IP, exact match on IP and range

match on port respectively in a pre-defined topology. Figure 2 shows one of the available profiles, which is used by VNET-to-VNET to implement CA-PA mapping policy.

4.4 Flow Actions

We define a set of flow actions for DASH pipeline to modify packets of stateful flows. These actions comprehensively cover the packet transformations used by our cloud, including VxLAN & NVGRE encapsulation, MAC & IP rewriting, SNAT & DNAT and IPv4-IPv6 translation. Flow actions are independent, meaning that the fields that a flow action modifies do not overlap, and are executed centrally by `Action Apply` stage for both slow path and fast path packets; prior stages only publish actions to metadata bus without touching the actual packet. Such designs are intended to avoid the inconsistency caused by direct packet modifications by prior stages, and to simplify and optimize hardware implementation, *e.g.,* reducing latency through the parallel execution of independent actions.

5 SmartSwitch: A Uni-box Hardware Model

Our second contribution is SmartSwitch — the new uni-box hardware model that directly integrates DPUs with NPU, providing scalable performance and memory space for states with high space-, power- and cost-efficiency. In this section, we present its designs and rationales, and explain why we prefer it over other hardware models to serve DASH pipelines.

5.1 Design Overview

Architecture. We depict the architecture of SmartSwitch in Figure 1: a SmartSwitch is built by integrating an NPU with multiple DPUs via interconnects, *e.g.,* backplane Ethernet or PCIe, leveraging the strengths of both components. The NPU delivers packets intended for service processing to the backend DPUs. DPUs implement DASH pipelines and maintain flow states. SmartSwitch combines the deployability of NPU-centric model with the scalable performance and state of DPU-centric model. Compared to DPU-centric Sirius, DPUs of a SmartSwitch process packets directly on the data-path rather than off-path with extra routing hops, which reduces latency and jitter; merging the roles of datacenter switches and DPU appliances can also save costs due to the elimination of redundant equipment, improve physical scalability by re-using wiring, powering, cooling facilities and rack space of existing switches, and allow for the scale-out of stateful processing capability coupled with the number of server racks.
Deployment positioning in Azure. We model SmartSwitch as MoR (T1) switches instead of T0 or T2 switches for current workloads in Azure. This ensures that a SmartSwitch can be reached by any VM in the same datacenter and allows flexible placement of services across resource pools, providing the same disaggregation benefits as Sirius's. There are also enough redundant links between any pair of T1 SmartSwitches (either via T0 and T2), and high availability (HA) of services can be achieved with "stand-by" copy and inline/bulk sync techniques from our previous works [30, 32] using these redundant links, as opposed to the possible loss of flow states and service interruptions due to lack of interconnectivity when SmartSwitches deployed at T0 or T2 fail.

5.2 Handling Underlay and Overlay Traffic

The central challenge of designing SmartSwitch is, how to split the stateful cloud service workload between NPU and DPU to fully leverage their distinct strengths? Note that since we merge the roles of switches and network service appliances, a SmartSwitch needs to process both underlay traffic, *e.g.,* east-west forwarding, and overlay traffic, *i.e.,* stateful traffic intended for offloaded services. We choose to use SmartSwitch NPU to exclusively handle underlay traffic due to its high bandwidth (O(10Tbps)), and its limited on-chip resources are sufficient for storing underlay forwarding tables.

For stateful traffic, we choose to offload the entire DASH pipeline including both slow and fast path to SmartSwitch DPUs. Overlay traffic is distinguished from underlay traffic and forwarded to DPUs via VM-SmartSwitch tunnels: the SmartSwitch advertises a virtual data-path IP (VIP) through BGP to attract stateful traffic. Azure VMs whose network services are offloaded should establish a tunnel to the SmartSwitch by encapsulating their packets with hardcoded, reserved VNI and SmartSwitch VIP (as outer DIP). The NPU can then distinguish overlay traffic from underlay traffic with the tunneling header and forward overlay traffic to DPUs.

Our design choice to offload both fast and slow path to DPUs originates from their distinct processing patterns:
DASH pipeline fast path. Fast path conducts look-ups on flow table, and requires a large amount of flows (O(10M)) and their states to be stored in the data plane to ensure fast look-ups for high throughput and low latency. Modern DPUs, *e.g.,* Pensando DPU used by Sirius, are equipped with ASIC pipelines with significantly larger SRAM/DRAM than NPU's, and are therefore better at supporting fast path processing of O(10M) flows at line rate.
DASH pipeline slow path. Slow path modifies the states that are to be accessed by fast path, *e.g.,* entry creation, deletion, and update of flow states. Due to architectural limitations, it is non-trivial to support inline modification operations with ASICs, *e.g.,* P4 pipeline, on commodity NPUs and DPUs [31, 32, 45]. On-board CPUs of switches and DPUs can flexibly modify data plane states from the control plane. Although a single core can only support these operations at ∼100s Kop/s, we can leverage the unique advantage of SmartSwitch, wherein the number of backend DPUs can be adjusted, to achieve scale-up and scale-down of slow path capability. Slow path can also leverage low-latency on-board interconnects (NoC) to achieve high operations per second if it is co-located with fast path states on DPUs.

Therefore, we currently offload both paths on SmartSwitch DPUs and let the NPU focus on underlay traffic. DPUs are

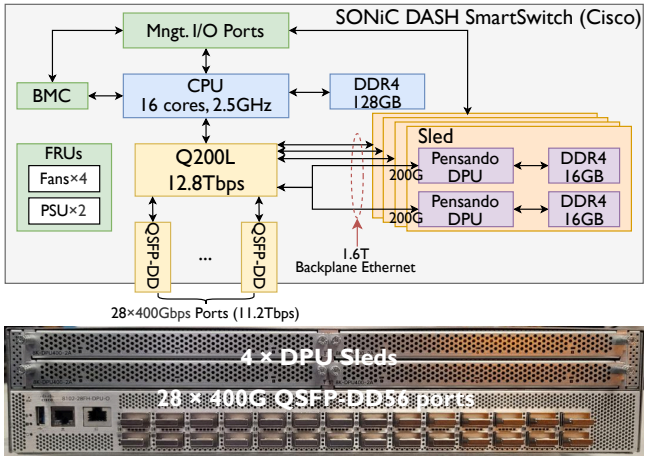

Figure 3: Block diagram and front-panel photo of SONiC DASH SmartSwitch (Cisco) [11]. Front panel: 28×400G QSFP-DD56 network-facing ports; chassis supports up to 8 Pensando DPUs (4 field-replaceable sleds).

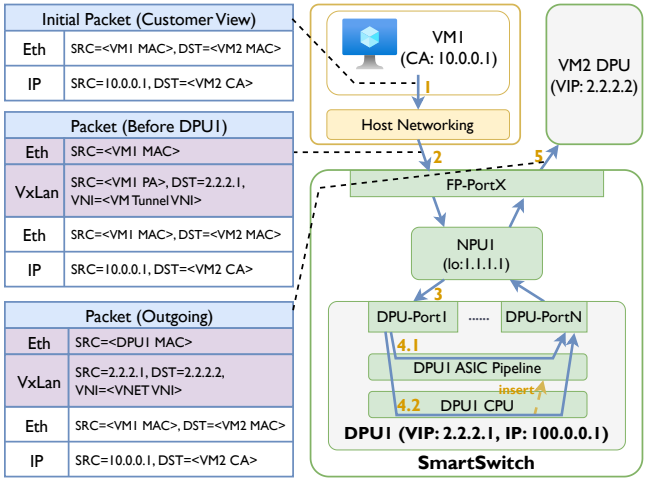

Figure 4: Packet path and transformations of VNET-to-VNET outbound on DASH SmartSwitch.

loosely coupled with NPUs, and we are able to flexibly scale-up and scale-down stateful processing resources according to varying loads in different regions by simply adjusting the number of installed DPUs of a SmartSwitch and trade-off between provisioned capacity and cost. Our current design achieves a total throughput, concurrent flows and CPS that equals the number of DPUs per SmartSwitch multiplied by the throughput, concurrent flows and CPS of a single DPU, wherein the number of DPUs is limited by the form factor of the SmartSwitch. For example, the SmartSwitch hardware implemented by Cisco, shown in Figure 3, allows 8×200Gbps Pensando DPUs per NPU, achieving 1.6Tbps throughput, 256M concurrent connections and 20M new connections/s for stateful overlay traffic, and 11.2Tbps for underlay traffic (4 out of 32×400Gbps ports are used for connecting DPUs internally) simultaneously.

Packet processing path. Azure VMs are purchased with virtual NICs (*a.k.a.* Elastic Network Interfaces or ENIs) that provide network services of certain capacity. Figure 4 shows the example packet processing path with SmartSwitch offloads when two VMs communicate in an Azure virtual network through the VNET-to-VNET services provided by their ENIs.

(1) VMs communicate with each other using customer addresses (CA), which are private virtual IP addresses in virtual networks. Therefore, the source and destination addresses of the initial packet are VM1's and VM2's CAs.

(2) The host NIC performs a stateless encapsulation with a reserved VNI and VIP (*e.g.,* `2.2.2.1`) to tunnel the packet to the SmartSwitch that offloads VM1's ENI.

(3) SmartSwitch NPU identifies the traffic encapsulated with the reserved VNI as stateful traffic, and uses the data path VIP and ENI MAC (inner src MAC for VNET-to-VNET) to locate the DPU. If the DPU is local,[5] NPU directly forwards the packet to the local DPU by moving the packet to the DPU's ingress port. Here we choose to use the ENI-based forwarding approach on NPU, *i.e.,* looking up the next-hop DPU based on VIP + ENI MAC instead of VIP. The design rationales are: allocating a unique VIP for each SmartSwitch DPU is expensive considering the future scale of SmartSwitch deployments, and the physical placement of ENIs should be flexible and transparent for user VMs. ENI-based forwarding can be generalized to any services that require isolation between tenant pipelines by using other fields as identifiers in the `Pipeline-Lookup` stage.

(4) After the packet enters the DPU that offloads VM1's ENI, it is processed stage by stage as programmed with DASH pipeline. For packets that trigger a flow miss in `ConnTrack-Lookup`, *e.g.,* TCP SYN or the first packet of a UDP flow, they are trapped by `ConnTrack-Update` to *data plane app* (4.2), which is a run-to-completion software packet-processing engine on CPU. Data plane app uniformly manages the interaction between the CPU and ASICs of a DPU via DASH-SAI APIs. For flow creation, data-plane app invokes the DASH API `create_flow_entry` to modify ASIC states, then re-injects the packet to ASIC through veth interfaces for remaining stages. Following packets of created flows will be directly processed by DPU ASICs (4.1).

(5) The outgoing packet is encapsulated with the VIP of VM1's and VM2's SmartSwitch as SIP and DIP and VNET's VNI, and routed to VM2's ENI.

6 Community-Driven Development Model

Our third contribution is a community-driven development model for SONiC DASH SmartSwitch. This model follows a fundamentally different principle from the vendor-centric model: Instead of letting vendors dictate which features are available for cloud usage, we invite both cloud providers and hardware vendors to collectively maintain a vendor-neutral agreement as a community. The agreement reflects community consensus on what features are required, *i.e.,* what stages

[5] Due to page limits, we show the path when a DPU is remote (NPU-NPU tunnel) in Appendix §C.

should be included in DASH pipeline and what APIs should be included in DASH.

6.1 Challenges

We have several practical considerations when designing the agreement:

Q1. How do we ensure that DASH pipeline behaviors are reproducible with hardware from any vendor? How do we ensure that vendors can understand and interpret DASH pipeline behavior without ambiguity?

Q2. How do we know that vendors can 100% implement what we specify within DASH pipeline? For example, is adjusting valid fields in flow matching from 5-tuple to 4-tuple achievable without making hardware or code changes?

Q3. How can we translate DASH pipeline specifications to DASH APIs and ensure that those APIs conform to SAI's data models? Can we do this with an automated CI workflow instead of handcrafting APIs details?

Q4. How can we ensure backward-compatibility when a new feature is introduced to DASH pipeline?

Since DASH is an extension of SAI, a strawman solution is to directly follow its agreement form, *i.e.,* documentation written in natural language. However, it is unsuitable for DASH due to inherent impreciseness and significant manual efforts required to handle API details according to SAI's data model.

6.2 P4 as the Single Source of Unambiguous and Executable Truth

Based on the above considerations and design objective III (§2), we decide to use P4 to model the service behavior as a white box, which we refer to as *behavior model*. Although any formal language, *e.g.,* C and Python, can avoid ambiguity when modeling the behavior of DASH pipeline, P4 has several advantages and can effectively resolve the practical concerns (§6.1) we have for the agreement form. For Q1 and Q2, P4-programmable hardware has been widely deployed. We therefore use P4 to balance between service requirements and hardware capability: it acts as a "lower bar" that ensures DASH pipeline can be implemented on a majority of hardware platforms from major hardware technology providers and as an "upper bound" that ensures we (and other cloud providers in the community) do not design stages, attributes and behaviors of DASH pipeline so freely that they end up being hard to implement with hardware (as they must be representable with P4). Note that we acknowledge that P4 semantics and architectures vary across backends (e.g., software BMv2 and TNA). However, the goal of the behavior model (based on V1model) is not to serve as a deployable codebase for all hardware, but to act as a "Golden Model" of DASH pipeline behavior. Vendors are expected to create their own implementations — whether using target-specific P4, microcode, or RTL — that are functionally equivalent to this model. For Q3 and Q4, the behavior model is executable code by itself and enables new cloud requirements to be timely

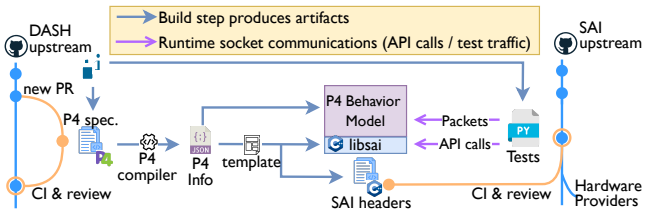

Figure 5: Community-driven development model for DASH.

absorbed into DASH ecosystem via automated continuous integration (CI) workflow. With the mature tool-chain of P4, *e.g.,* compilers and software backend, the behavior model can be compiled for: (1) automatically generating SAI-compatible control APIs, which is a more efficient and less error-prone way to extend SAI with new stateful processing features compared to handcrafting, and (2) running on a software switch, which allows test cases of new features to be written and validated before SmartSwitch hardware roll-outs and accelerates the development process (*a.k.a.* "shift-left" testing).

6.2.1 Auto-Generating Control API

We generate SAI-compatible DASH APIs from the P4 specifications of DASH pipeline automatically. This is feasible because both P4 specifications and SAI APIs are structured, making it easy to translate. Note that when adding a new feature, its corresponding new DASH APIs are also created in the exact same way. The generation process can be done in two steps as shown in Figure 5. First, we leverage the open-source P4 compiler to compile the specification into P4 Info, which is a structured intermediate representation in JSON format with information of all P4 entities, including P4 tables, actions, counters and types. Then an API generator we build renders pre-defined templates into DASH-SAI API headers. Specifically, the generated artifacts include CRUD operation prototypes (create, remove, set and get) for P4 tables and related data structures, *e.g.,* state entry, strictly under SAI's data model and standards.

6.2.2 Shift-Left Testing

Before SmartSwitches from any hardware technology provider enters Azure, it is necessary to conduct conformance and performance tests. Conformance tests validate whether the hardware implementation of DASH pipeline meets the expected behaviors defined in P4 specifications. For performance tests, we define "hero tests targets" [22], *i.e.,* the production-ready performance and policy scale of Azure services, as concrete references that hardware vendors can target when optimizing their designs. For these two kinds of tests, we leverage the executable nature of P4 sources to achieve *shift-left testing* and further disaggregate cloud service software (programming and testing) and hardware implementation. We borrow BMv2 [21, 28], which is an open-source software switch, to load compiled P4 Info and run DASH pipeline. Since P4 sources faithfully model the behavior of DASH pipeline, cloud providers can write and run tests against BMv2 before real switch hardware actually becomes available, and

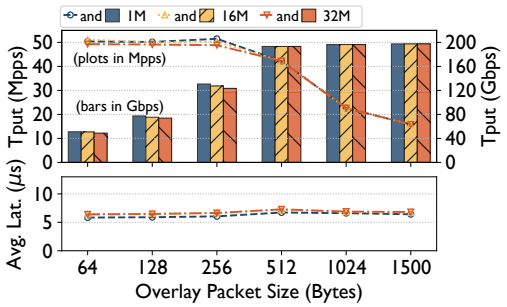

Figure 6: Throughput of SmartSwitch with one backend DPU and end-to-end latency of packets under different number of concurrent flows.

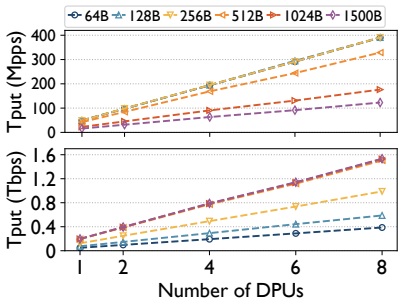

Figure 7: Throughput of SmartSwitch with different number of DPUs. Each DPU has 32M concurrent flows and even traffic.

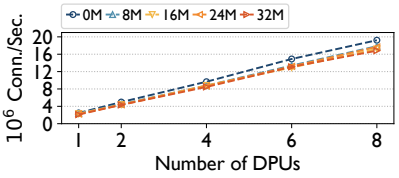

Figure 8: CPS achieved by SmartSwitch with different number of DPUs under different number of background flows per DPU. Background flows are UDP flows at 0.1 Mpps/flow with 64-byte packet. Requests are evenly distributed to all DPUs.

reuse the same set of tests afterwards, thereby "shifting-left" testing of services on their develop-test-deploy timeline.

Since vanilla BMv2 uses P4Runtime as its control plane, test cases written against it cannot be applied to non-P4 hardware, which defeats the goal of shift-left testing. As shown in Figure 5, we address this by auto-generating BMv2-specific implementations of SAI APIs (`libsai` library), which serve as the adapter that translates SAI API table and attribute CRUD operations into equivalent P4Runtime RPC calls. With `libsai`, shift-left test cases can be directly applied to heterogeneous hardware implementations (with SmartSwitch as one of them) due to the consistency of SAI APIs across targets.

7 Evaluation

In this section, we comprehensively evaluate SONiC DASH SmartSwitch from three aspects, including performance, efficiency and multi-sourcing benefits.

7.1 Methodology

Testbed. We use the Cisco 8102-28FH-DPU-O SmartSwitch shown in Figure 3 for stress benchmarking. The switch is equipped with a 12.8T Q200L Silicon One NPU [13] for packet forwarding and a 16-core x86 CPU with 128GB DRAM for running the SONiC operating system. The switch has four port-facing, field-replaceable DPU sleds, each of which supports two 200G AMD Pensando Elba DPUs [3], for a total of 8 DPUs supporting up to 1.6T of DPU services. Each DPU is equipped with 16 ARM CPU cores and 32GB of memory. To evaluate the SmartSwitch performance in the datacenter environment, we use an IXIA traffic generator equipped with an Ultra High-Density (UHD) device to generate full line-rate, VxLAN-encapsulated traffic.

Service. We configure the VNET-to-VNET service on Cisco SmartSwitch, which fully exercises all stages that are currently supported by DASH pipeline and is therefore ideal for benchmarking. The policy scale is shown in Appendix §E.

Traffic and methodology. For throughput and latency benchmarking, we apply bi-directional VxLAN-encapsulated UDP flows and adjust the number of concurrent flows by changing the number of source IPs and UDP source ports. We configure the traffic generator to generate traffic that is uniformly distributed over all flow entries on DPUs. This choice is informed by our Azure live site incidents, which reveal that performance degradations — particularly those that are notoriously difficult to diagnose in production (e.g., silent packet drops and latency bumps) — are primarily caused by bottlenecks in the DPU/switch memory hierarchy when handling high-frequency access to large-scale flow tables. We therefore use the traffic pattern that can maximize the pressure applied on the memory hierarchy of SmartSwitch DPUs and ensure that high throughput and low latency can be sustained under any traffic pattern. For CPS benchmarking, we use pairs of HTTP servers and clients to initiate short-lived TCP connections with different source ports.

7.2 Performance

Throughput and latency benchmarking. We first install one DPU in a sled of the SmartSwitch to evaluate fast-path packet processing performance. Figure 6 shows the throughput in Mpps (plots) and Gbps (bars), and the average packet-processing latency of SmartSwitch. SmartSwitch with a single DPU easily achieves 32M concurrent flows, and exhibits high throughput and stable latency across different packet sizes and concurrent flows in fast-path processing. For example, at 32M connections, the SmartSwitch achieves 49.3Mpps, 49.1Mpps, and 48.8Mpps for packet sizes of 64B, 128B, and 256B, respectively, fully saturating the packet processing capability (pps limit) of the backend DPU. For larger packet sizes of 512B, 1024B, and 1500B, it achieves 193.2Gbps, 196.4Gbps, and 197.5Gbps, respectively, fully utilizing the bandwidth capacity (bps limit) of the backend DPU. The same behavior is observed when the number of concurrent flows is less than 32M, satisfying the service requirements on both concurrent connections and throughput in our cloud. For latency, SmartSwitch shows relatively stable average latency (5.8μs to 7.2μs), in which NPU processing takes 1.0μs.

Scale-up to multiple DPUs. A key benefit of our SmartSwitch model is that the number of DPUs can be flexibly adjusted to scale-up resources and performance. To demonstrate this, we further increase the number of installed DPUs on the SmartSwitch to 2, 4, 6, 8 and measure the throughput. Figure 7 shows that the throughput of fast-path processing can be

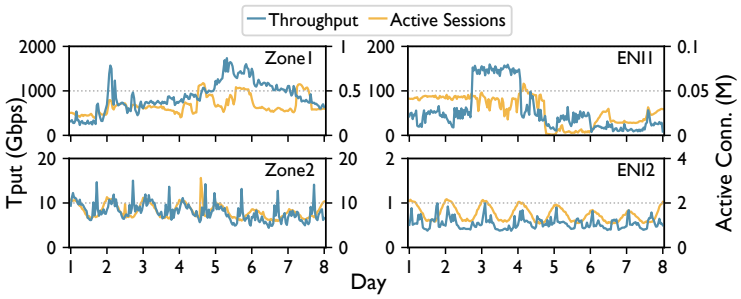

Figure 9: Production results of throughput and number of active sessions collected from Azure during Q2.

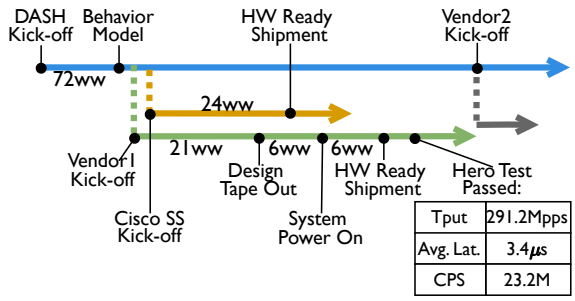

Figure 10: Timeline of multi-sourced development of SONiC DASH SmartSwitch, "ww" refers to work week.

linearly scaled to 391.02Mpps for small packets (64B, 128B, 256B) and 1.53Tbps for large packets (512B, 1024B, 1500B) when 8 DPUs are installed. The SmartSwitch NPU can forward stateful traffic to corresponding ENIs on DPUs at full line rate and does not become a bottleneck.

CPS benchmarking. Figure 8 presents the CPS achieved by Cisco SmartSwitch with different numbers of DPUs under varying numbers of background UDP flows, which may affect the CPS due to different flow table sizes. Without background flows, the SmartSwitch achieves linear scaling of connection rates from 2.47M CPS (1 DPU) to 19.22M CPS (8 DPUs) under the HTTP traffic we use, indicating ultra-high performance. For scenarios where background connections exist, connection rate still increases linearly with the number of DPUs, while more background connections lead to slightly lower CPS due to more frequent hash collisions in flow tables. For example, when the flow table stores 32M background UDP flow entries per DPU, CPS of the full-fledged SmartSwitch (8 DPUs) drops to 16.82M.

Production deployment. We deploy SmartSwitches in Azure and report the throughput and the number of active connections of two Azure Zones (AZ) and two ENIs collected over a week in Figure 9. We observe that, due to different VM workloads, AZs and ENIs exhibit different service traffic patterns and demands, *e.g.,* Zone1 exhibits high bandwidth while Zone2 exhibits a high and periodic number of active connections respectively. We also observe that these two metrics of some highly demanding ENIs, *e.g.,* ENI1 and ENI2 selected from Azure, can reach ∼160Gbps and ∼2M during peak hours. Deployed SmartSwitches can support both average and peak loads from customers with a reasonable amount of capacity reserved for the organic growth of new ENIs.

7.3 Deployment Footprints and Efficiency

SmartSwitch consumes 958.83W and 1040.83W of power at 0% and 100% PPS load, which is 365.82W lower than the power consumption of Sirius. In addition to absolute power consumption, we quantify "efficiency" as *the throughput achieved by per unit of power, cost and rack space*, and "footprint" as *the additional power and rack space used by a solution compared to existing datacenters*, and compare these metrics of SmartSwitch and Sirius. Results presented in Table 6 (Appendix §F) show that SmartSwitch achieves 1.81×, 2.68× and 1.33× higher power-, space- and cost-efficiency than Sirius. This can be explained by reduced auxiliary equipment that makes no contribution to higher BPS, PPS, CPS and number of flows. For footprints, since SmartSwitches are built upon or replace previous T1 switches, each SmartSwitch only consumes 1 unit of additional rack space and 448.72W of additional power with the 8 DPUs it equips, and does not require additional wiring, which results in better scalability under physical constraints compared to Sirius.

7.4 Multi-Sourcing Analysis

Figure 10 briefly shows the multi-sourced development process of SmartSwitches since the launch of DASH project. By specifying our service requirements with DASH behavior model in a one-time effort, we have saved ∼144 work-weeks for software development so far, which would otherwise have been used for developing services on different vendors' platforms. The development of DASH is also accelerated by collective efforts from the community (10-25 PRs per week). For SmartSwitch hardware, multiple vendors can work in parallel. The Cisco SmartSwitch is delivered with a 6-month lead time; the SmartSwitch from another vendor (Vendor1) also reaches hero performance targets and is delivered with a 9-month lead time as shown in Figure 10. We expect the benefits of multi-sourcing will continue to grow with new vendors, *e.g.,* anonymous Vendor2, joining the DASH community.

8 Experiences

SONiC DASH SmartSwitch was initiated in 2021, and since then 841 community members have contributed to the open-source codebase [14]. In cooperation with 13 hardware technology providers, we have already deployed SONiC DASH SmartSwitch in Azure and are continuing its rollout, from which we have learned several important lessons worth sharing. Beyond the lessons discussed earlier (§2), we have also gained the following insights from operating SmartSwitch.

Reduce hard downtime with rolling updates. Firmware updates of network devices require device reboot and can cause "hard downtime", during which related network services become completely non-functional. When we deploy SmartSwitches in Azure and adopt the same update approach

as regular switches', *i.e.,* whole-device reboot, the hard downtime becomes unacceptable because (1) SmartSwitches are newly designed and integrated hardware components for our clouds and we need to frequently apply updates to a large fleet to resolve system bugs, introduce new DASH features and improve performance, and (2) a SmartSwitch can serve large amounts of VMs who will be affected by updates. We leverage the separated DPUs of SmartSwitch and apply "rolling" updates on DPUs to address this, *i.e.,* updating the DPUs one by one, which only temporarily affects the system's total capacity without causing complete service disruption. We call this "soft downtime" in contrast to hard downtime and use rolling updates to trade off between the two downtimes for different workloads and SLA. On a related note, our workload assignment for NPU also minimizes hard downtime because updating NPU will block the entry point for SmartSwitch traffic, but since the NPU only focuses on simple functions such as packet forwarding by our design, the need for frequently updating it is minimized.

Use T0 for bulky flow synchronization. As we have discussed in §5, we adopt solutions in Microsoft's prior works [30, 32] to achieve HA for service flow states on SmartSwitch. However, during production deployments, we identified a new and unaddressed issue on path selection for bulky flow synchronization between SmartSwitches (for quick replication or migration of flow states from one SmartSwitch to another). SmartSwitches are deployed at the T1 layer and lack direct connections to other SmartSwitches at the same layer, so synchronization must occur through either T0 or T2 layer. On one hand, the T2 layer often exhibits a high oversubscription ratio. On the other hand, T0 switches typically reside in the same failure domain, sharing the same fate. Thus, both design choices have inherent limitations. We prefer using T0 to route bulk sync traffic in production deployments because using T2 switches is more likely to cause network congestion; additionally, the probability of multiple T0 switches failing simultaneously has become very low in modern datacenters.

Fine-grained debugging for stateful workloads. We observe several scenarios wherein fine-grained debugging is highly required on SmartSwitches in production deployments and discuss two of them here. The first scenario is: we need to dump the flow table to check policy correctness. Due to the extraordinarily high CPS and flow scale on SmartSwitches, we found that a full dump of a flow state table is time-consuming (*e.g.,* 32M entries can take tens of seconds), with flow states potentially changing during the operation, leading to outdated results. We address this by introducing fine-grained filters in DASH-flow APIs to retrieve only specified flows. The second scenario is: we want to verify whether packets hit all pipeline stages correctly in all cases. We developed DASH `Packet Hit Test` APIs that can specify which packets or flows are of interest, and return the matched entries and metadata as these packets traverse each stage of the DASH pipeline, facilitating fine-grained debugging based on real-time ASIC states.

9 Related Work

In addition to our own prior efforts listed in Table 1, SONiC DASH SmartSwitch is related to the following works. (1) SilkRoad [41] offloads stateful load balancers to programmable switches. Sailfish [44] leverages programmable switches to offload cloud gateways and is deployed in Alibaba's cloud. These works can only support a subset of all flows due to memory limitations. (2) Later works build hybrid NPU/DPU/FPGA/server systems to overcome the resource limitations of a single programmable device. TEA [36] uses memory on remote servers to store states and assist programmable switches. Tiara [45] leverages FPGA-based SmartNICs and servers to assist programmable switches and build a scalable architecture for L4 load balancers. ExoPlane [37] pairs server DPUs with ToR Tofinos to store large numbers of flow states. These works rely on specific proprietary hardware and interfaces, incurring high costs and deployment footprints. (3) We share a similar vision on the importance of physical deployability as [42] but we extend the scope to other factors, *i.e.,* programming and development models. Luoshen [43] builds a highly-converged 2U appliance with Tofino, FPGA and x86 cores to reduce footprints in edge, but it suffers from high OPEX and vendor lock-in, and cannot support the high number of flows in public clouds. Albatross [40] mitigates the supply risks of Sailfish and Luoshen by adopting commodity x86 cores and FPGAs, but it leaves the problem of the vendor-centric development model unaddressed. Nezha [38] reuses existing SmartNICs without introducing new devices and reduces CAPEX, but it has high latency overhead (up to 10μs) due to remote processing. In summary, these works are promising but only address specific deployability concerns and have performance issues compared to our approach.

10 Conclusion and Future Direction

We share experiences and problems with the programming model, hardware model and development model of existing cloud network service solutions in Azure. Based on the experiences, we design, build, and deploy SONiC DASH SmartSwitch and accomplish design goals of high performance, physical deployability, and fast feature iteration simultaneously. Future directions of SONiC DASH SmartSwitch include NPU-DPU co-processing and extending DASH to new use cases, and we sincerely invite new members to join the community and make contributions.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Vishal Shrivastav, for their constructive feedback. We also thank all contributors in the DASH community for their valuable efforts. We further thank Keysight Technologies, especially Swami Balasubramanian and Mircea Dan Gheorghe, for their strong support. Shaofeng and Hong are supported in part by funding from The Chinese University of Hong Kong (4937007, 4937008, 5501329, 5501517).

References

[1] Alibaba Cloud. https://www.alibabacloud.com.

[2] Alibaba Cloud: NAT Gateway. https://www.alibabacloud.com/help/en/nat-gateway/.

[3] AMD Pensando DPU Technology. https://www.amd.com/en/products/data-processing-units/pensando.html.

[4] AWS. https://aws.amazon.com.

[5] Azure Load Balancer. https://aka.ms/AA101zg8.

[6] Azure NAT Gateway. https://aka.ms/AA101zga.

[7] Azure Private Endpoint. https://aka.ms/AA101zg7.

[8] Azure Private Link. https://aka.ms/AA101rw9.

[9] Azure Virtual Network. https://aka.ms/AA101zgb.

[10] Azure Virtual Network Peering. https://aka.ms/AA101zg6.

[11] Building High-Performance Network Services with Cisco Smart Switch: Microsoft Success Story. https://www.ciscolive.com/c/dam/r/ciscolive/global-event/docs/2024/pdf/CSSSPG-1015.pdf.

[12] Cisco 8101-32FH. https://www.router-switch.com/8101-32fh.html.

[13] Cisco Silicon One Q200 and Q200L Processors Data Sheet. https://www.cisco.com/c/en/us/solutions/collateral/silicon-one/datasheet-c78-744312.html.

[14] DASH. https://github.com/sonic-net/DASH.

[15] Dell PowerEdge R940. https://newserverlife.com/server-models/dell-poweredge-r940.

[16] Google Cloud. https://cloud.google.com/.

[17] Intel Tofino 2, 6.4 Tbps, 4 Pipelines. https://www.intel.com/content/www/us/en/products/sku/218647/intel-tofino-2-6-4-tbps-4-pipelines/specifications.html.

[18] NVIDIA Bluefield-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[19] NVIDIA Bluefield-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf.

[20] Open Compute Project. https://github.com/opencomputeproject.

[21] p4lang/behavioral-model: The reference P4 software switch. https://github.com/p4lang/behavioral-model.

[22] Policy and Route Requirements. https://github.com/sonic-net/DASH/tree/main/documentation/general/program-scale-testing-requirements.

[23] RFC 7348 - Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. https://datatracker.ietf.org/doc/rfc7348.

[24] RFC 7637 - NVGRE: Network Virtualization Using Generic Routing Encapsulation. https://datatracker.ietf.org/doc/rfc7637.

[25] SAI. https://github.com/opencomputeproject/SAI.

[26] SONiC. https://github.com/sonic-net/SONiC.

[27] Tencent Cloud. https://cloud.tencent.com.

[28] V1 model. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4.

[29] Manikandan Arumugam, Deepak Bansal, Navdeep Bhatia, James Boerner, Simon Capper, Changhoon Kim, Sarah McClure, Neeraj Motwani, Ranga Narasimhan, Urvish Panchal, Tommaso Pimpo, Ariff Premji, Pranjal Shrivastava, and Rishabh Tewari. Bluebird: High-performance SDN for bare-metal cloud services. In *Proc. USENIX NSDI*, 2022.

[30] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating stateful network functions. In *Proc. USENIX NSDI*, 2023.

[31] Tommaso Caiazzi, Mariano Scazzariello, and Marco Chiesa. Millions of low-latency state insertions on asic switches. *Proc. ACM Netw.*, 1(CoNEXT3), 2023.

[32] Ying Chu, Ziyuan Liu, Riff Jiang, Ze Gan, Junhua Zhai, Guohan Lu, Zhixiong Niu, and Yongqiang Xiong. FHA: Flow-level high availability on programmable network hardware for cloud provider. In *Proc. ACM APSys*, 2024.

[33] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc De Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. USENIX NSDI*, 2018.

[34] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *Proc. USENIX NSDI*, 2017.

[35] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the public cloud. In *Proc. USENIX NSDI*, 2018.

[36] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: Enabling state-intensive network functions on programmable switches. In *Proc. ACM SIGCOMM*, 2020.

[37] Daehyeok Kim, Vyas Sekar, and Srinivasan Seshan. ExoPlane: An operating system for on-rack switch resource augmentation. In *Proc. USENIX NSDI*, 2023.

[38] Xing Li, Enge Song, Bowen Yang, Tian Pan, Ye Yang, Qiang Fu, Yang Song, Yilong Lv, Zikang Chen, Jianyuan Lu, Shize Zhang, Xiaoqing Sun, Rong Wen, Xionglie Wei, Biao Lyu, Zhigang Zong, Qinming He, and Shunmin Zhu. Nezha: SmartNIC-based virtual switch load sharing. In *Proc. ACM SIGCOMM*, 2025.

[39] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. ServerSwitch: A programmable and high performance platform for data center networks. In *Proc. USENIX NSDI*, 2011.

[40] Jianyuan Lu, Shunmin Zhu, Jun Liang, Yuxiang Lin, Tian Pan, Yisong Qiao, Yang Song, Wenqiang Su, Yixin Xie, Yanqiang Li, Enge Song, Shize Zhang, Xiaoqing Sun, Rong Wen, Xionglie Wei, Biao Lyu, and Xing Li. Albatross: A containerized cloud gateway platform with fpga-accelerated packet-level load balancing. In *Proc. ACM SIGCOMM*, 2025.

[41] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. ACM SIGCOMM*, 2017.

[42] Jeffrey C. Mogul and John Wilkes. Physical deployability matters. In *Proc. ACM HotNets*, 2023.

[43] Tian Pan, Kun Liu, Xionglie Wei, Yisong Qiao, Jun Hu, Zhiguo Li, Jun Liang, Tiesheng Cheng, Wenqiang Su, Jie Lu, Yuke Hong, Zhengzhong Wang, Zhi Xu, Chongjing Dai, Peiqiao Wang, Xuetao Jia, Jianyuan Lu, Enge Song, Jun Zeng, Biao Lyu, Ennan Zhai, Jiao Zhang, Tao Huang, Dennis Cai, and Shunmin Zhu. LuoShen: A hyper-converged programmable gateway for multi-tenant multi-service edge clouds. In *Proc. USENIX NSDI*, 2024.

[44] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proc. ACM SIGCOMM*, 2021.

[45] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *Proc. USENIX NSDI*, 2022.

Appendix

A Acronyms and Their Definitions

Acronym	Explanation
NPU	Network Processing Unit, a programmable switch chip
DPU	Data Processing Unit
ToR	Top of rack, T0
MoR	Middle of rack, T1
SAI	Switch Abstraction Interface
SONiC	Software for Open Networking in the Cloud
DASH	Disaggregated APIs for SONiC Hosts, used interchangeably in this paper with DASH-SAI APIs
MAT	Match-action table
5-tuple	<source IP, destination IP, source port, destination port, protocol>
VNI	Virtual network identifier, *e.g.,* VxLAN ID or GRE key
CA	Customer Address, address visible inside customer VM
PA	Platform Address, Physical Address, Provider Address, internal datacenter address used for routing
ENI	Elastic Network Interface, a virtual NIC of the user VM with a unique MAC address
VIP	Virtual data-path IP (VIP) of SmartSwitch
Sled	A modular, field-replaceable module that plugs into a chassis or enclosure and provides standardized electrical, mechanical, and management interfaces (power, network, and control) for devices such as DPUs, CPUs, or storage.
AZ	Azure Zone, a logical set of one or a few Azure data centers that are geographically adjacent

Table 3: General and Azure-specific acronyms (in the order of first presence).

B DASH Pipeline Stages

Stages supported in the current version of DASH is shown in Table 4.

C Packet Path for Remote SmartSwitch DPU

Figure 11 shows the packet path of VNET-to-VNET if the ENI of VM1 is on a remote SmartSwitch DPU. NPU of SmartSwitch 2 can check inner source MAC address of packets to identify whether the ENI is local. Packets for remote DPUs will be transmitted via a NPU-NPU tunnel.

D P4 Specifications and DASH-SAI APIs

The P4 specification of `ConnTrack-Lookup` stage in DASH pipeline is shown in Figure 12. Generated SAI APIs based on this P4 specification are called DASH-Flow APIs and are shown in Figure 12. To provide APIs for manipulating entries in the flow table, we explicitly set `isobject` to true for `flow_table` and leave `isobject` as false for `flow_entry` in the P4 source. The effect is: the name "flow_table" is borrowed as the name of a SAI object and the name "flow_entry" is borrowed as the name of a SAI object's attribute, and CRUD APIs of both will be generated accordingly (with different format for an object and an attribute) as shown in Figure 12. Hardware technology providers should take care of the underlying implementation of `ConnTrack Lookup` stage and DASH-Flow APIs to ensure correct data-plane and control-plane behavior by referencing P4 sources, generated APIs and SAI data model. For the two APIs shown in Figure 12, this means hardware technology providers should ensure that `sai_create_flow_table_fn` creates a flow table, which is a SAI object, and `sai_create_flow_entry_fn` should insert a flow entry, which is an attribute of the flow table object, to flow table on their SmartSwitch devices.

Stage	Functionality
Packet Parsing	Extracting packet header information, *e.g.,* source and destination MAC, and populate corresponding fields in metadata bus.
Direction Lookup	Inbound and outbound direction identification for processing bi-directional flows correctly.
Pipeline Lookup	Directing traffic to a pipeline based on configured pipeline identifier.
Packet Decap	Decapsulating all outer encapsulation of a packet.
ConnTrack Lookup	Flow entry lookup. Flow-miss will cause packet to enter slow path of DASH pipeline, while flow-hit will publish flow action information into metadata bus.
Pre-pipeline ACL	Applying admission control policies and drop unexpected packets before Matching Stages.
Matching Stages	Matching packets based on specific fields for metadata publishing and customized policy enforcement.
Post-pipeline ACL	Applying admission control policies and drop unexpected packets after Matching Stages.
ConnTrack Update	Inserting, deleting and updating flow entries according to policies set by Matching Stages.
Action Apply	Applying all packet transformation actions (stored in metadata bus) specified by previous stages, *e.g.,* ConnTrack Lookup stage.
Underlay Routing	Setting next hop of the packet based on the destination IP address (outer destination IP).
Meter Update	Updating the metering counters if any metering class is specified.
Packet Out	Emitting packets.

Table 4: Supported stages in DASH pipeline and their functionalities.

E Experiment Configuration

The specific configurations that we use for VNET-to-VNET in §7 are listed in Table 5.

Metrics	Value
VNIs	32
ENIs	32
ACL rules	320000
VNETs (Outbound)	32
Inbound Routes	32
Outbound routes	160K
CA-to-PA mapping (Outbound)	80320

Table 5: Policy scale of VNET-to-VNET.

F Efficiency Calculations

A complete Sirius appliance includes a pair of 3U servers equipped with 6 AMD Pensando DPUs each for redundancy and reliability, and 2 additional 1U switches together with optics for connectivity with datacenter network. For a fair comparison, we only calculate power, space and cost of half of a complete appliance without redundancy. Power consumption statistics of a SmartSwitch, a Pensando DPU and a Sirius

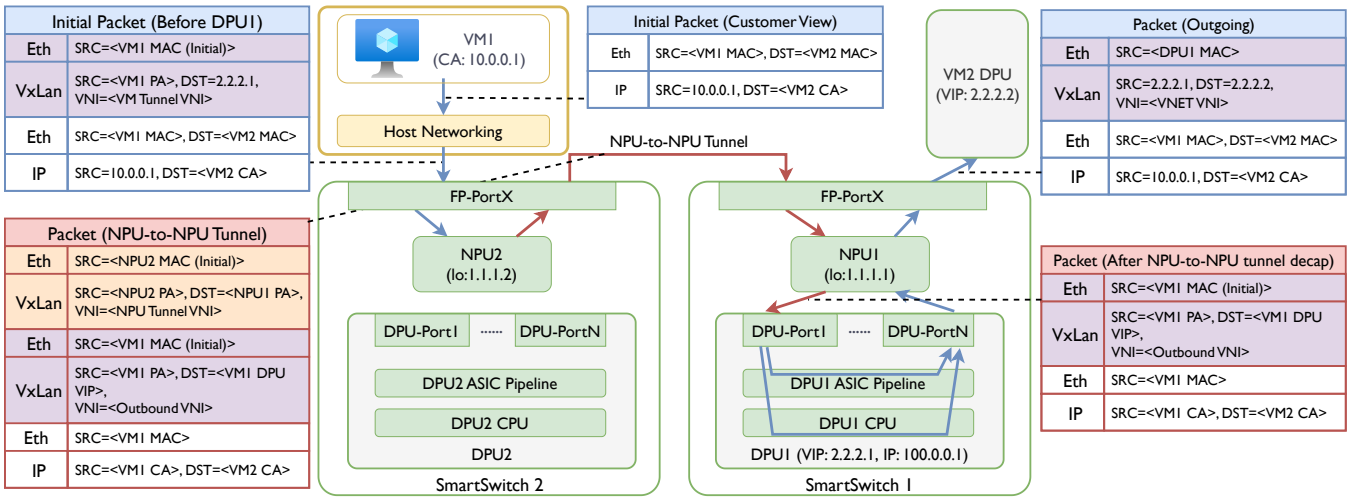

Figure 11: Packet path of VNET-to-VNET for remote DPU scenario.

conntrack_lookup.p4

```
control conntrack_lookup_stage(inout headers_t hdr, inout metadata_t meta){
    action set_flow_table_attr(dash_flow_enabled_key_t dash_flow_enabled_key, ...){
        meta.flow_table.flow_enabled_key = dash_flow_enabled_key;
        // Set other flow table attributes
        }
    @SaiTable[name="flow_table", api="dash_flow", order=0, isobject= "true"]
    table flow_table{
        key = {meta.flow_table.id:exact;}
        actions = {set_flow_table_attr;}
    }

    action set_flow_entry_attr(...){...}
    action flow_miss(...){...}
    @SaiTable[name="flow", api="dash_flow", order=1, enable_bulk_get_api="true", \
    enable_bulk_get_server="true"] // Directives for SAI API generation.
    table flow_entry{
        key = {
            meta.flow_key.eni_mac:exact;
            meta.flow_key.vnet_id:exact;
            meta.flow_key.src_ip:exact;
            meta.flow_key.dst_ip:exact;
            meta.flow_key.src_port:exact;
            meta.flow_key.dst_port:exact;
            meta.flow_key.ip_proto:exact;
        }
        actions = {
            set_flow_entry_attr; // Action for publishing metadata for matched flow.
            @defaultonly flow_miss;
        }
        const default_action = flow_miss;
    }
    apply{
        flow_table.apply();
        if (meta.flow_table.flow_enabled_key & dash_flow_enabled_key_t.ENI_MAC != 0)
        {
           hdr.flow_key.eni_mac = meta.eni_addr; // Mask valid flow key fields.
        }
        ... // Other flow key fields are set similarly.
        flow_entry.apply();
    }
}
```

saiexperimentaldashflow.h

```
typedef struct _sai_flow_entry_t
{
    sai_object_id_t switch_id;
    sai_mac_t eni_mac;
    sai_uint16_t vnet_id;
    sai_uint8_t ip_proto;
    sai_ip_address_t src_ip;
    sai_ip_address_t dst_ip;
    sai_uint16_t src_port;
    sai_uint16_t dst_port;
} sai_flow_entry_t;

/**
 * @brief Create flow entry
 * @param[in] flow_entry Entry
 * @param[in] attr_count Number of attributes
 * @param[in] attr_list Array of attributes
 * @return #SAI_STATUS_SUCCESS on success Failure
 *         status code on error
 */
typedef sai_status_t (*sai_create_flow_entry_fn)(
        _In_ const sai_flow_entry_t *flow_entry,
        _In_ uint32_t attr_count,
        _In_ const sai_attribute_t *attr_list);
// Other APIs for flow entry...
typedef sai_status_t (*sai_create_flow_table_fn)(
        _Out_ sai_object_id_t *flow_table_id,
        _In_ sai_object_id_t switch_id,
        _In_ uint32_t attr_count,
        _In_ const sai_attribute_t *attr_list);
// Other APIs for flow table...
```

saidashflow.py

```
sai_thrift_create_flow_table()
sai_thrift_create_flow_entry()
...
send_packet()
verify_packet()
```

Figure 12: P4 specification source of `ConnTrack Lookup` stage and generated SAI header (DASH-Flow APIs). Tests written against DASH APIs can be used for testing any DASH-capable hardware target.

server are collected from PSU, sensors and BMC respectively under 100% PPS load of 64-byte packets. Additional power and space (footprints) are calculated against datacenter infrastructure without Sirius and SmartSwitch, *i.e.,* with only normal T1 switches. Prices are based on open sources [12,15] since we cannot disclose pricing from hardware technology providers publicly. PPS per CPU core of server is obtained from [34].

Metric	Sirius[1]	SmartSwitch
Power [2] ↓ (W)	1406.65	**1040.83**
Space ↓ (RU)	4	**2**
Additional Power [3] ↓ (RU)	1406.65	**448.72**
Additional Space ↓ (RU)	4	**1**
Power Efficiency ↑ (Mpps/kW)	207.11	**374.76**
Space Efficiency ↑ (Mpps/RU)	72.83	**195.03**
Cost Efficiency[4] ↑ (Mpps/K$)	2.80	**3.73**
Savings[5] ↑ (# cores)	530	**709**

Table 6: Efficiency comparisons. Better ones are highlighted.

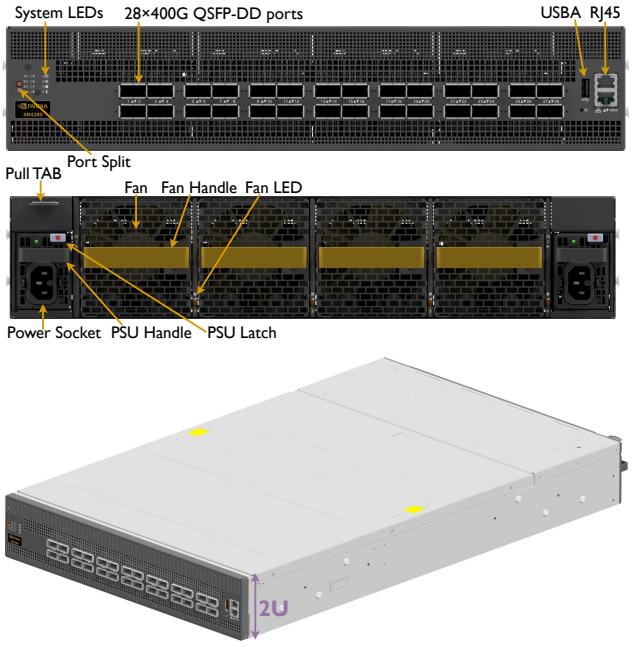

Figure 13: Nvidia's SONiC DASH SmartSwitch in front and rear view. The SmartSwitch supports Nvidia Bluefield-3 DPU and 28×400G QSFP-DD ports.

G Artifact Appendix

Abstract

SONiC-DASH is an open source project with the goal to "deliver enterprise network performance to critical cloud applications". The project extends functionality to stateful workloads. DASH has joined the Linux Foundation to participate in the large communities of developers and users contributing there, and also collaborates with Open Compute Project (OCP) [20].

The DASH repository includes high-level documentation, DASH pipeline behavior model (P4 sources), test plan and test cases, DASH-SAI headers and other related artifacts of DASH. Together with open-sourced SONiC, they form the software stack of SONiC DASH SmartSwitch.

Scope

- For new community members and contributors: help you quickly understand the high level design of DASH, including DASH overall architecture, DASH service scenarios in Azure, DASH-SONiC integration, DASH functional and hero tests, *etc.*, with rich documentation.
- For contributors:
  - provide you with the source code of DASH pipeline behavior model (P4 sources), compilation toolchain and containerized environment to assist with the development of new stages and to run and validate the pipeline on the reference software switch (BMv2).
  - provide you with the source code of DASH test cases to assist with the development of new test cases and comprehensively validate the compliance of new stages.
  - provide you with the toolchain for automatically generating DASH-SAI headers for new stages, which conform to SAI standards and can be merged as part of SAI.
- For hardware technology providers:
  - provide you with the behavior model of DASH pipeline as the groundtruth for hardware implementations to align with.
  - provide you with tests that can be directly applied on DASH-capable hardware to verify conformance and performance.
- For the community: for the first time, provide the community, including both cloud providers and hardware providers, with a venue and a development paradigm for discussing service requirements and defining cloud network services in a vendor-neutral manner.

Contents

We list the content that is highly relevant and of high significance in DASH and SONiC repositories.

- sonic-net/dash/: DASH
  - dash-pipeline/
    - SAI/: SAI submodule and templates.
    - bmv2/: behavior model implementation of DASH pipeline, written in $P4_{16}$ with BMv2 v1model [21]. The current version of DASH pipeline supports 13 stages.
    - dpapp/: source code of data-plane app.
    - tests/: libsai and saithrift API tests. CI-ready functional tests for DASH pipeline are in /test.
    - README-dash-workflows.md: quick start guide on the developer workflow (building and running BMv2, running tests).
  - documentation/: high-level design documents, baseline specifications and requirements for DASH-capable devices, service specifications and requirements (*e.g.,* VNET-to-VNET) for DASH-capable devices and others.
  - slides/: slides for SONiC DASH SmartSwitch related talks.
  - test/: CI-ready functional and hero tests for DASH pipeline, test configuration generation tools and other

related artifacts.

- sonic-net/SONiC/doc/
  - dash/: detailed design document of DASH APIs, DASH orchestration agent, Config and APP DB Schemas and other SONiC buildimage changes required to bring up SONiC image on DASH-capable devices.
  - smart-switch/: high-level design documents for SONiC DASH SmartSwitch, including BFD, HA, database architecture and others.

Hosting

DASH is hosted on GitHub as part of SONiC foundation. The latest version of DASH can be accessed at the main branch.

DASH-SAI APIs

The SAI project (hosted as a standalone project on Github) is absorbing DASH APIs as experimental features [25]. 17 groups of APIs in total have been merged into SAI by the time the final paper is ready. These APIs are defined with C headers, which are automatically generated with the P4-to-DASH and P4-to-`libsai` translation templates written in Jinja. Specifically, 6814 lines of DASH APIs (4810 lines of comments and 2004 LoC) have been automatically generated.

Supported SmartSwitch Hardware

We collaborate with 12 hardware technology providers to design and build SmartSwitch in a community-driven manner. Cisco and Nvidia have introduced their DASH-capable SmartSwitch hardware (Figure 3 and 13) based on the designs of SmartSwitch hardware model (§5). These SmartSwitches run SONiC and fully support DASH pipeline and DASH APIs on both their NPU and DPU.

Community Contributions

DASH is community-driven. Over 4 years, the Github repository has accumulated 108 contributors, with 822 commits made and 425 pull requests merged to the main branch by the time the final paper is ready. The broader community has attracted hundreds of contributors across Microsoft, AMD, Cisco, Nvidia, Keysight and others, enabling multi-vendor collaboration for hyperscale cloud networking.